Memory Management Simulator
Design Document

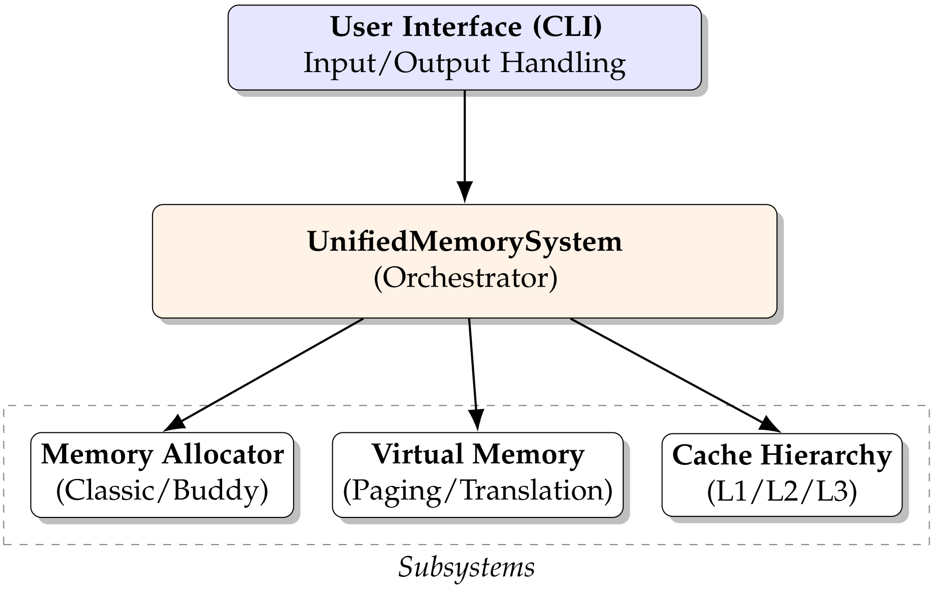
Sneha Kandpal
23116092
Electronics and Communication Engineering

Contents

1 System Overview

1.1 Architecture

The simulator implements a unified memory management system that models three key OS subsystems. The architecture is hierarchical, with the `UnifiedMemorySystem` acting as the orchestrator.



1.2 Design Principles

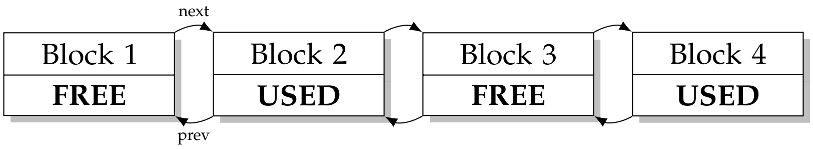
1. **Modularity**: Each subsystem (allocator, cache, VM) is independent.
2. **Configurability**: Users can enable/disable components as needed.
3. **Accuracy**: Algorithms faithfully simulate OS behavior.
4. **Transparency**: Verbose mode shows internal operations.
5. **Extensibility**: Easy to add new policies and features.

2 Memory Allocator Design

2.1 Classic Allocator

2.1.1 Memory Representation

Memory is modeled as a **doubly-linked list** of memory blocks.



2.1.2 Data Structure: MemoryBlock

MemoryBlock Struct

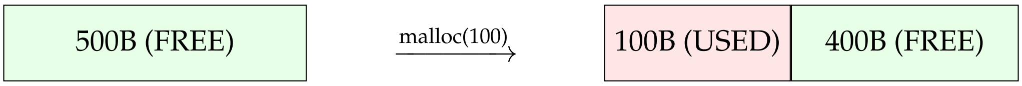
```
struct MemoryBlock {
    size_t start_address;  // Starting byte address
    size_t size;           // Block size in bytes
    bool is_allocated;     // Allocation status
    int block_id;          // Unique identifier
    MemoryBlock* next;     // Next block in list
    MemoryBlock* prev;     // Previous block in list
};
```

2.1.3 Allocation Strategies

- **First Fit**:
  - Traverses list from beginning; allocates first block that fits.
  - **Time Complexity:** $O(n)$.
  - **Advantage:** Fast allocation.
  - **Disavantage:** Can cause fragmentation at list start.
- **Best Fit**:
  - Searches entire list for smallest sufficient block.
  - **Time Complexity:** $O(n)$.
  - **Advantage:** Minimizes wasted space.
  - **Disavantage:** Slower, creates tiny unusable fragments.
- **Worst Fit**:
  - Selects largest available block.
  - **Time Complexity:** $O(n)$.
  - **Advantage:** Leaves large holes.
  - **Disavantage:** Quickly fragments large blocks.

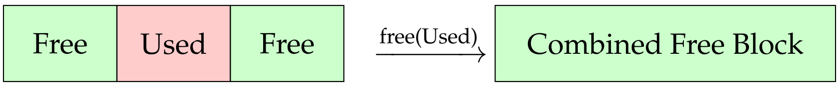
2.1.4 Block Splitting

When allocated block is larger than requested:



2.1.5 Coalescing

Merges adjacent free blocks to reduce fragmentation.



Algorithm: After freeing a block, check `prev` and `next`.

1. If `prev` is free: merge current into `prev`.
2. If `next` is free: merge `next` into current.
3. Update pointers to maintain list integrity.

3 Buddy System Design

3.1 Overview

Buddy allocation uses **power-of-2** sized blocks organized in free lists.

3.2 Memory Organization

Memory is divided into orders where each order represents a specific block size:

Order 0: 16B	[●][●][●] . . .	(`min_block_size`)
Order 1: 32B	[●][●] . . .	
Order 2: 64B	[●][●] . . .	
. . .		
Order N: Total Memory	[●]	(`max_order`)

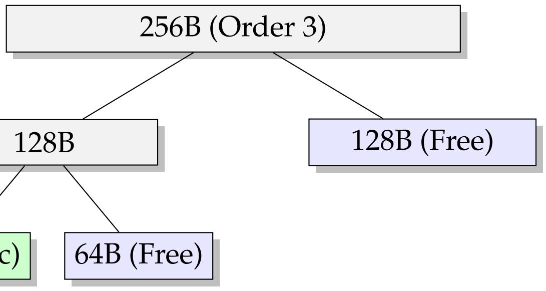
3.3 Data Structures

Code Snippet

```
struct BuddyBlock {
    size_t address;        // Block starting address
    size_t size;           // Block size (power of 2)
    bool is_free;          // Availability
    int block_id;          // Allocation ID
    BuddyBlock* next;      // Next in free list
};
vector<BuddyBlock*> free_lists;  // One list per order
map<int, AllocationRecord> allocated_blocks;   // Track allocations
```

3.4 Block Splitting

When a block is requested (e.g., 64B) and only larger blocks are available (e.g., 256B), the system performs recursive splitting.

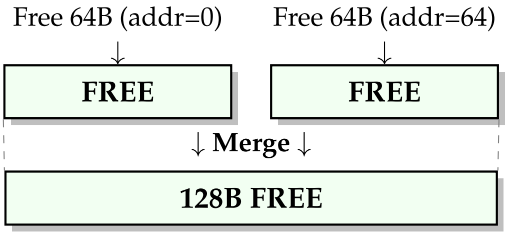


Algorithm:

1. If no block exists at the desired order, recursively split a larger block.
2. Create two "buddy" blocks of size/2.
3. Add both buddies to the free list of the next lower order ($order - 1$).
4. Repeat until the desired size is reached.

3.5 Buddy Merging

When two buddies are free, they merge back up the tree to form a larger block.



Buddy Address Calculation: To find the address of a block's buddy, we toggle the bit corresponding to the block size:

$$\text{buddy\_addr} = \text{address} \oplus \text{size} \tag{1}$$

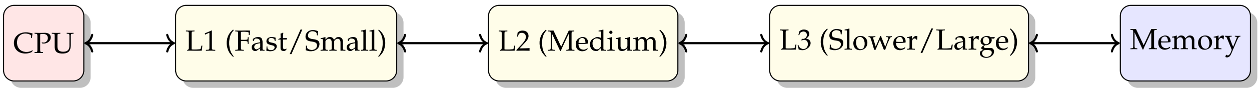
3.6 Internal Fragmentation

Because allocations must be powers of 2, rounding up causes wasted space (Internal Fragmentation). For example:

- **Request:** 100B
- **Allocated:** 128B (Next power of 2)
- **Waste:** $128B - 100B =$ **28B** (Internal Fragmentation)

4 Cache Hierarchy Design

4.1 Multi-Level Organization

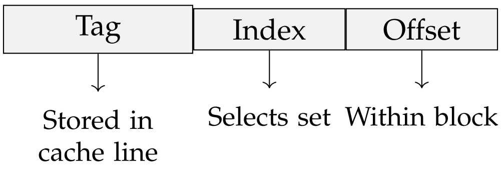


4.2 Cache Line Structure

```
struct CacheLine {
    bool valid;           // Is entry valid?
    size_t tag;           // Address tag
    bool dirty;           // Modified (for write-back)
    int insertion_order;  // For FIFO
    int last_access_time; // For LRU
};
```

4.3 Address Decomposition

For a memory address:



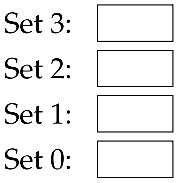
Calculations:

```
block_number = address / block_size;
set_index = block_number % num_sets;
tag = block_number / num_sets;
```

4.4 Associativity

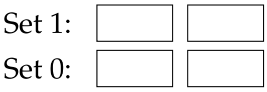
Direct-Mapped (1-way)

Each address maps to exactly one line.



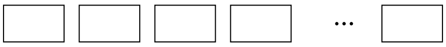
2-Way Set Associative

Each address can go in 2 lines.



Fully Associative

Address can go anywhere.

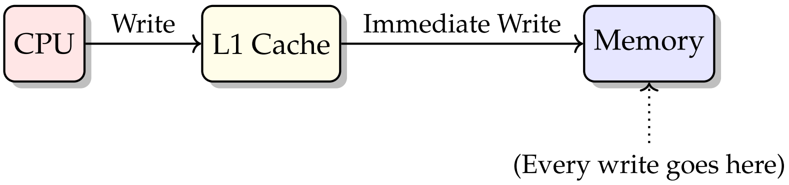


4.5 Replacement Policies

- **FIFO (First-In-First-Out)**:
  - Track insertion order.
  - Evict oldest entry.
  - Simple, but ignores access patterns.
- **LRU (Least Recently Used)**:
  - Track last access time.
  - Evict least recently used.
  - Better hit rate, more complex.

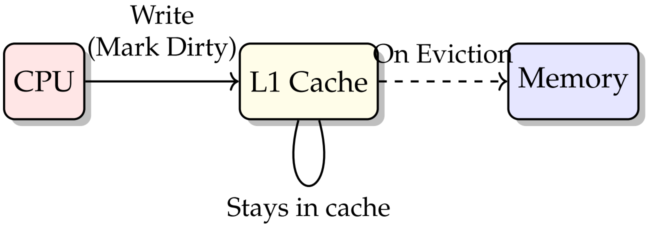
4.6 Write Policies

4.6.1 Write-Through



- **Pros**: Memory always consistent.
- **Cons**: High memory traffic, slower.

4.6.2 Write-Back



- **Pros**: Fewer memory writes, faster.
- **Cons**: Complex, stale memory until eviction.

4.6.3 Dirty Bit Tracking

```
[Valid=1] [Tag=15] [Data=...]  [Dirty=1]
```

The `Dirty=1` indicates the data is modified but not yet written to memory.

When evicting a dirty block:

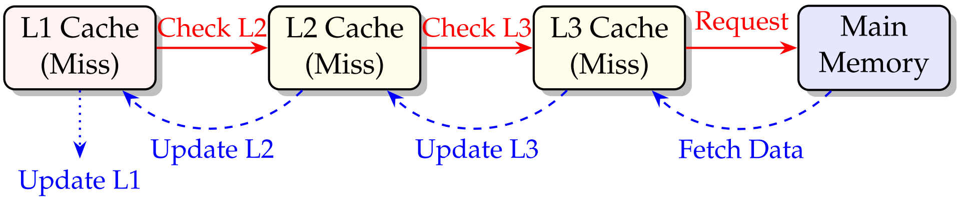
1. Write block contents to memory (write-back).
2. Increment write-back counter.
3. Replace with new block.

4.7 Write-Allocate Policy

The simulator implements Write-Allocate for write misses. When writing to an address not present in cache, the system fetches the block from memory into cache before performing the write, ensuring modified data is available for subsequent accesses, leveraging temporal locality.

4.8 Hierarchy Behavior

Read Miss Flow:



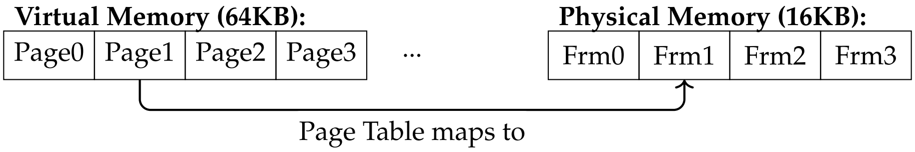
Write Miss Flow (Write-Allocate):

1. **L1 Write Miss** → Fetch block (check L2, L3, Memory).
2. Insert into L1.
3. Perform write (mark dirty if write-back).

5 Virtual Memory Design

5.1 Paging System

Virtual memory is divided into fixed-size **pages**, physical memory into **frames**.



5.2 Page Table Entry

```
struct PageTableEntry {
    bool valid;           // Page in memory?
    int frame_number;     // Physical frame (-1 if not in memory)
    bool dirty;           // Page modified?
    int last_access_time; // For LRU
    int load_time;        // For FIFO
    int access_count;     // Statistics
};
```

5.3 Address Translation

Example Calculation:

- **Virtual Address:** 0x0518 (1304 decimal)
- **Page Size:** 256 bytes

$$\text{Page Number} = 1304/256 = \mathbf{5}$$
$$\text{Offset} = 1304\%256 = \mathbf{24}$$

Mapping: `Page Table[5]` → **Frame 2**

$$\text{Physical Address} = (2 \times 256) + 24 = \mathbf{536}$$

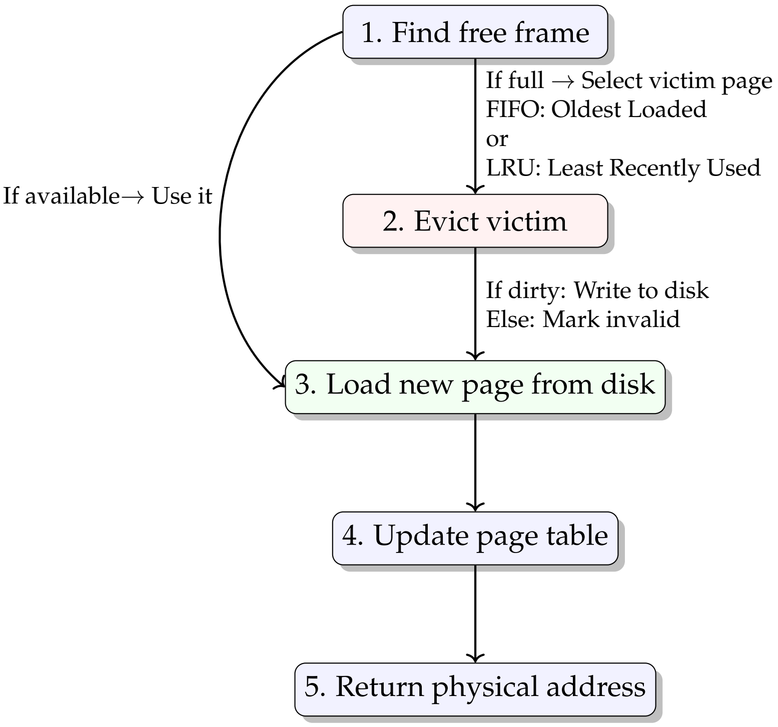
Algorithm:

```
page_number = virtual_address / page_size;
offset = virtual_address % page_size;

if (page_table[page_number].valid) {
    frame = page_table[page_number].frame_number;
    physical_address = (frame * page_size) + offset;
    return physical_address;  // Page HIT
} else {
    handlePageFault(page_number);  // Page MISS
}
```

5.4 Page Fault Handling

When accessing an unmapped page:



5.5 Page Replacement Policies

FIFO (First-In-First-Out)

Frames:	P0	P1	P2	P3
	Load time: 10	Load time: 15	Load time: 20	Load time: 25

Evict oldest (P0)

LRU (Least Recently Used)

Frames:	P0	P1	P2	P3
	Access time: 10	Access time: 50	Access time: 45	Access time: 55

Evict LRU (P0)

6 Integration Architecture

6.1 Unified Flow

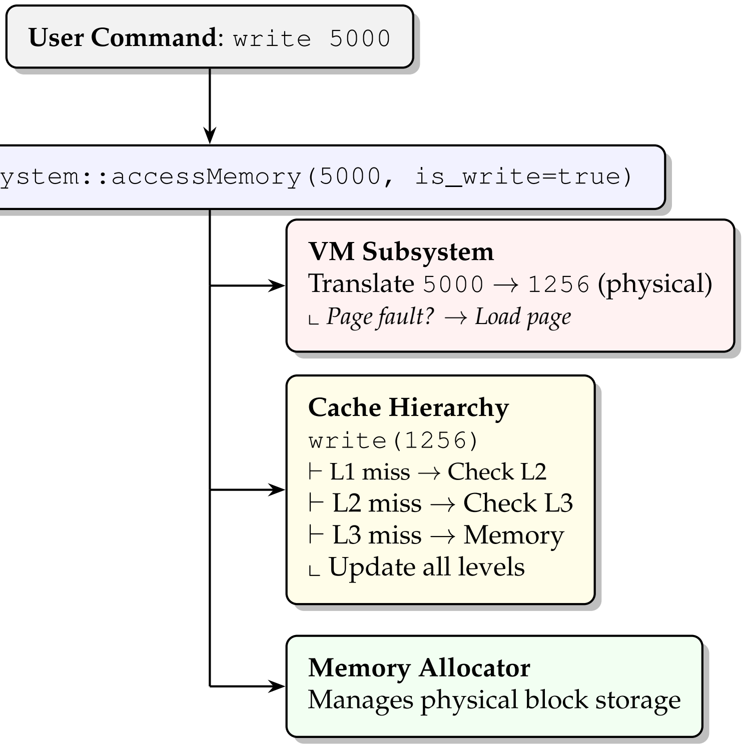
Access Memory Flow

```
void accessMemory(address, is_write) {
    // Step 1: Virtual Memory (if enabled)
    if (vm_enabled) {
        physical_addr = vm->translateAddress(address);
        if (physical_addr == INVALID) return;  // Translation failed
    } else {
        physical_addr = address;  // Direct addressing
    }

    // Step 2: Cache Hierarchy (if enabled)
    if (cache_enabled) {
        if (is_write) {
            cache->write(physical_addr);
        } else {
            cache->read(physical_addr);
        }
    }

    // Step 3: Physical Memory (implicit, managed by allocator)
    // Actual data access would happen here in real system
}
```

6.2 Component Interaction



7 Data Structures

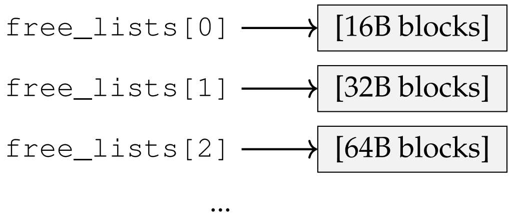
7.1 Memory Allocator

Doubly-Linked List: Allows efficient coalescing in both directions.

- **Advantage**: $O(1)$ merge with adjacent blocks.
- **Disadvantage**: $O(n)$ search for allocation.

7.2 Buddy System

Array of Free Lists: One list per block size.



- **Advantage**: $O(1)$ access to blocks of specific size.
- **Disadvantage**: Internal fragmentation.

7.3 Cache

2D Vector: `cache[set][way]`

Code Snippet

```
vector<vector<CacheLine>> cache;
```

- **Advantage**: Direct indexing, $O(1)$ set access.
- **Disadvantage**: $O(\text{ways})$ search within set.

7.4 Virtual Memory

Vector of Page Table Entries: One entry per virtual page.

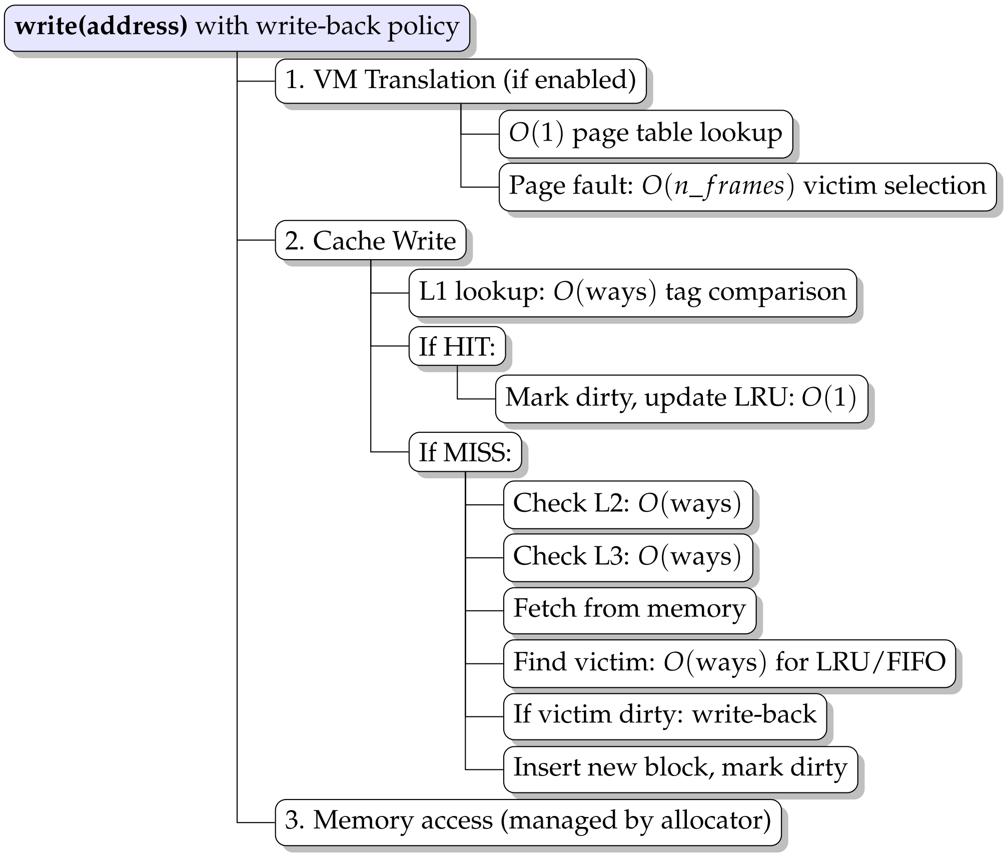
Code Snippet

```
vector<PageTableEntry> page_table;
```

- **Advantage**: $O(1)$ address translation.
- **Disadvantage**: Memory overhead for large address spaces.

8 Algorithms

8.1 Critical Path: Write Operation


write(address) with write-back policy
1. VM Translation (if enabled)
$O(1)$ page table lookup
Page fault: $O(n\_frames)$ victim selection
2. Cache Write
L1 lookup: $O(\text{ways})$ tag comparison
If HIT:
Mark dirty, update LRU: $O(1)$
If MISS:
Check L2: $O(\text{ways})$
Check L3: $O(\text{ways})$
Fetch from memory
Find victim: $O(\text{ways})$ for LRU/FIFO
If victim dirty: write-back
Insert new block, mark dirty
3. Memory access (managed by allocator)

8.2 Complexity Analysis

Operation	Time Complexity	Space Complexity
malloc (First Fit)	$O(n)$ blocks	$O(1)$
malloc (Best Fit)	$O(n)$ blocks	$O(1)$
free + coalesce	$O(1)$	$O(1)$
Buddy allocate	$O(\log n)$ splits	$O(n)$ free lists
Buddy free	$O(\log n)$ merges	$O(1)$
Cache access	$O(\text{associativity})$	$O(\text{sets} \times \text{ways})$
Page translation	$O(1)$	$O(\text{pages})$
Page fault	$O(\text{frames})$	$O(1)$

9 Design Decisions

9.1 Why Doubly-Linked List for Classic Allocator?

- **Decision**: Use doubly-linked list over array/bitmap.
- **Rationale**:
  - Allows $O(1)$ coalescing with both neighbors.
  - Dynamic sizing without pre-allocation.
  - Easy to split and merge blocks.
- **Trade-off**: $O(n)$ search vs. potential $O(\log n)$ with trees.

9.2 Why Separate Read and Write Methods in Cache?

- **Decision**: Implement `read()` and `write()` separately instead of generic `access()`.
- **Rationale**:
  - Write policies (write-through vs write-back) require different handling.
  - Dirty bit only applies to writes.
  - Allows separate statistics tracking.
  - Clearer semantics.
- **Trade-off**: More code, but more accurate simulation.

9.3 Why XOR for Buddy Address Calculation?

- **Decision**: Use `buddy_addr = address ^ size`.
- **Rationale**:
  - Elegant bit manipulation.
  - $O(1)$ computation.
  - Automatically finds buddy regardless of whether current block is left or right buddy.

Example:

```
Block at 0x0040 (64 decimal), size 64:
Buddy = 0x0040 ^ 64 = 0x0000 (left buddy)

Block at 0x0000, size 64:
Buddy = 0x0000 ^ 64 = 0x0040 (right buddy)
```

9.4 Why Two-Level Statistics (Cache-Level + Hierarchy-Level)?

- **Decision**: Track hits/misses at both individual cache level and hierarchy level.
- **Rationale**:
  - Cache-level: Shows performance of each cache.
  - Hierarchy-level: Shows overall system performance.
  - Allows analysis of inter-cache behavior.

- **Trade-off**: Slightly more complex bookkeeping.

9.5 Why Optional Components?

- **Decision**: Allow users to enable/disable VM and Cache independently.
- **Rationale**:
  - Educational: Study each subsystem in isolation.
  - Testing: Easier to debug individual components.
  - Flexibility: Support various scenarios.

10 Limitations

10.1 Simulator Limitations

1. **No Actual Data Storage**: Simulator tracks addresses and metadata, but doesn't store actual data bytes.
2. **Simplified Disk I/O**: Page faults don't actually read from disk, just simulate the operation.
3. **Single Process**: Only one "process" simulated at a time (one page table).
4. **No TLB**: Real systems use Translation Lookaside Buffer to cache page translations.
5. **Idealized Timing**: Miss penalties are fixed values, real hardware has variable latencies.
6. **No Prefetching**: Real caches often prefetch adjacent blocks.
7. **No Cache Coherency**: Multi-core cache coherency protocols not implemented.
8. **No Disk Simulation**: Virtual memory assumes infinite "disk" space.
9. **Fixed Block/Page Sizes**: Sizes are fixed per initialization cycle. However, the system supports runtime reconfiguration via the clear command, which allows the user to re-initialize subsystems with new parameters.

10.2 Design Simplifications

1. **Memory is Contiguous**: Assumes simple linear address space.
2. **No Memory Protection**: No checks for invalid accesses.
3. **Deterministic**: No randomization or non-determinism.
4. **User-Space Only**: No kernel/user mode distinction.
5. **No DMA**: Direct Memory Access not simulated.

10.3 Known Issue

Large Address Spaces: Very large VM configurations may use significant memory.

- **Mitigation**: Keep configurations reasonable ($<$ 1MB virtual memory).

11 Conclusion

This simulator provides an accurate, modular implementation of OS memory management concepts. The design prioritizes:

- **Educational Value**: Clear, understandable implementations.
- **Algorithmic Correctness**: Faithful to OS textbook algorithms.
- **Extensibility**: Easy to add new policies and features.
- **Usability**: Interactive CLI with helpful output.

While simplified compared to real hardware, the simulator captures the essential behavior and trade-offs of memory management systems.